M1 : Ingénierie du Logiciel – SORBONNE UNIVERSITE

Examen Réparti 2eme partie

17 Janvier 2019 (2 heures avec documents : tous SAUF ANNALES CORRIGEES). Barème indicatif sur 23 points (donne le poids relatif des questions)

Questions de cours [5 Pts]

Répondez de façon précise et concise aux questions.

Barème : VALABLE sur toutes les questions de cours : -25 à -50% si la réponse inclut la bonne idée, mais qu'elle est noyée dans des infos ou autres réponses fausses/inappropriées.

Question Cours (QC):

QC1) (1 point) Dans la méthode SCRUM, à quoi sert le « Backlog » et de quoi est-il constitué ?

QC2) (1 point) On considère des composants réalisant tous la même interface *IComponent* organisés sous la forme d'un arbre, suivant le DP Composite. Sur un diagramme de composant, représenter un composant *CComposite* interne (qui a des fils) et un composant *CLeaf* terminal (pas de fils).

QC3) (1 point) Définissez la notion de *Domain Specific Language (DSL)* et donner un exemple de DSL.

QC4) (1 point) Représentez et expliquez l'amélioration continue vue sous l'angle de la roue de Deming.

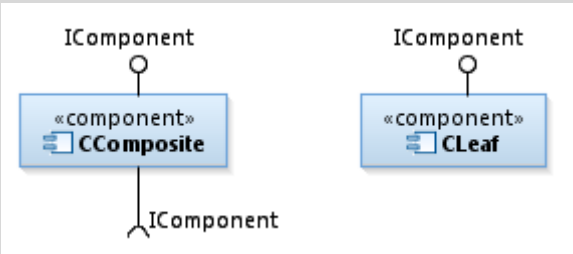
1)

C'est la liste des tâches à réaliser dans l'itération (ou au total selon la granularité). L'objectif de l'itération est donc de basculer tous les items du backlog de « TODO » à « DONE ». Chaque item décrit une tâche de développement, et peut être associé à une évaluation de difficulté ainsi qu'à une notion de criticité. On raffine progressivement le backlog au fil de l'itération. La célérité de l'équipe se mesure par la vitesse de traitement du backlog.

50% Nature : c'est un (sous) ensemble de tâches à réaliser au cours d'une itération pour satisfaire le besoin

50% Usage : On décrit au moins certaines de ses utilisations : on démarre l'itération il est plein, on le vide au cours du dév ; On s'en sert pour estimer la vitesse ; On s'en sert pour le suivi de la réalisation des besoins…

b)



CComposite : réalise **ET** utilise IComponent

CLeaf : réalise seulement IComponent

50% chacun

0% si la nature du diagramme est mauvais (e.g. dia de classe, de structure interne) Les diagrammes en plus de celui qui est demandé ne sont pas sanctionnés.

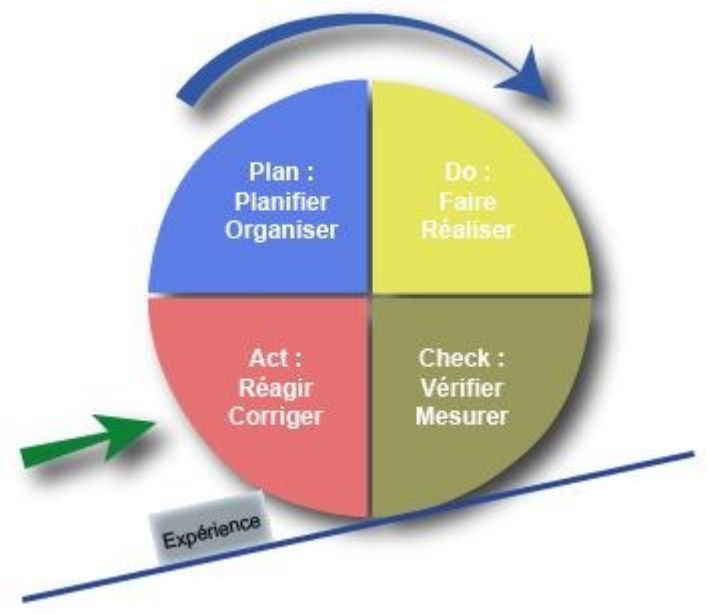
c)

70% c'est un « petit » langage, dédié à un domaine particulier, muni de peu de concepts, le plus souvent exécutable d'une manière ou d'une autre.

30% un exemple potable (il y a beaucoup de réponses possibles ici). En cours on a cité Makefile et SQL par exemple.

d)

Plan/Do/Check/Act + un bloqueur qui évite les régressions



50% on voit une roue qui progresse vers le « mieux » avec des segments nommés (étapes)

40% (10%*4) par étape de PDCA nommée,

10% bloqueur anti régression visible

2. Problème: Conception SUBOO [Barème sur 18 Pts]

Rappels d'analyse : Cet énoncé se base sur une version simplifiée de l'application SUBOO qui a fait l'objet du projet de TME. Nous considérons l'application SUBOO, permettant de générer des *build order* optimisés.

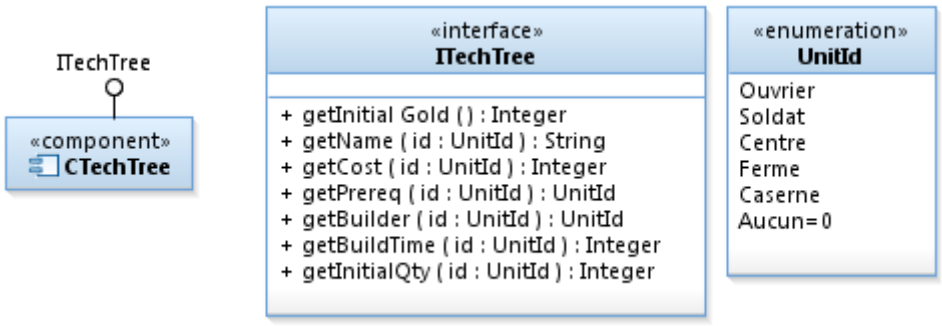
I. Composant CTechTree (5 points)

Les informations décrivant les unités sont présentées dans le cahier des charges sous la forme d'un tableau (NB : cette version est légèrement modifiée par rapport au projet de TME). Initialement chaque joueur dispose de 50 PO et les unités sont :

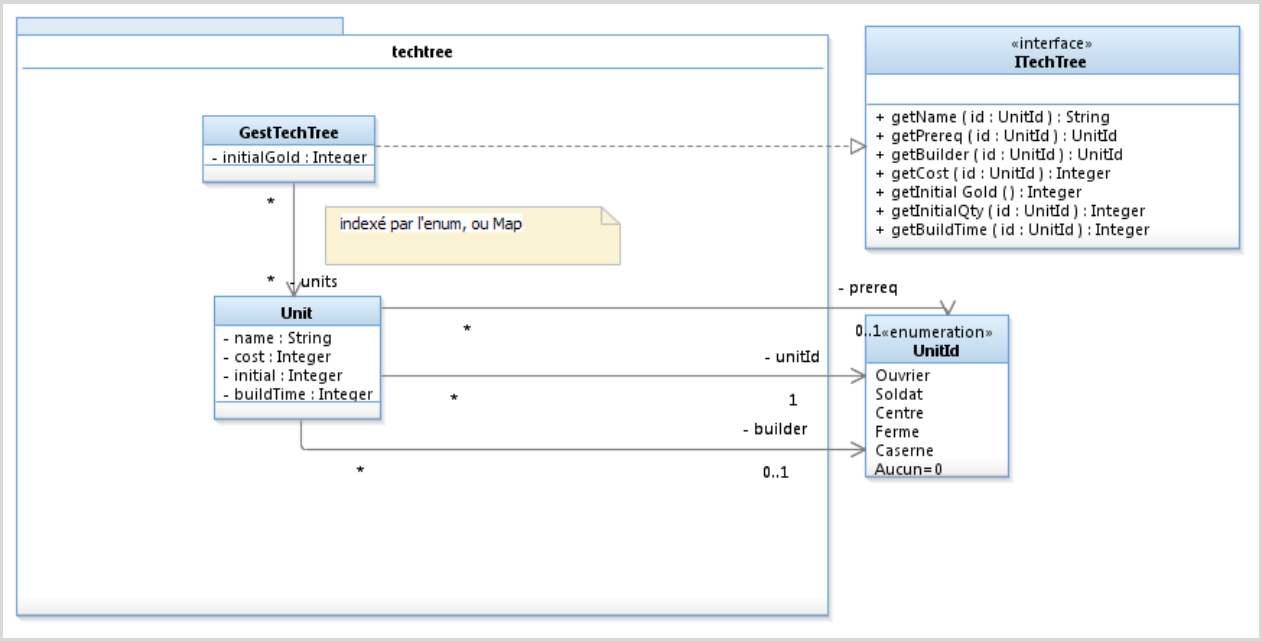
Unité	Cout PO	Prérequis	Construit par	Temps construction	Initial
Centre	300	Aucun	Ouvrier	200	1
Ferme	100	Aucun	Ouvrier	60	0
Caserne	150	Ferme	Ouvrier	100	0
Ouvrier	50	Aucun	Centre	30	5
Soldat	100	Aucun	Caserne	50	0

Ce composant bout de chaîne stocke donc les informations relatives aux unités disponibles dans le jeu et à l'état initial de la partie. On pourra faire varier sa réalisation si besoin pour traiter l'évolution des versions du jeu. Pour plus de lisibilité, on introduit une énumération pour identifier les types d'unités disponibles plutôt qu'un entier ou un String; il est donc possible pour un composant arbitraire de connaître

la liste d'unités disponibles en inspectant simplement l'énumération. On propose de réaliser le composant *CTechTree* suivant :



1. (1,5 point) Sur un diagramme de classes, donner une conception détaillée possible de ce composant *CTechTree*.



20% Gestionnaire réalise ITechTree

20% associe * Unit

10% gold sur Gestionnaire

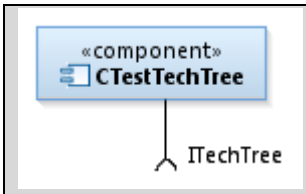
20% les 4 attributs : name, cost, temps, initial

30% les attributs prereq et builder, soit juste des UnitId comme dans le corrigé, soit des associations reflexives de Unit vers Unit.

-20% autres interfaces introduites

On souhaite mettre en place des tests d'intégration de ce composant.

2. (0,75 point) Représentez un ou plusieurs composants (testeur, bouchon(s)) permettant de mettre en place ces tests.



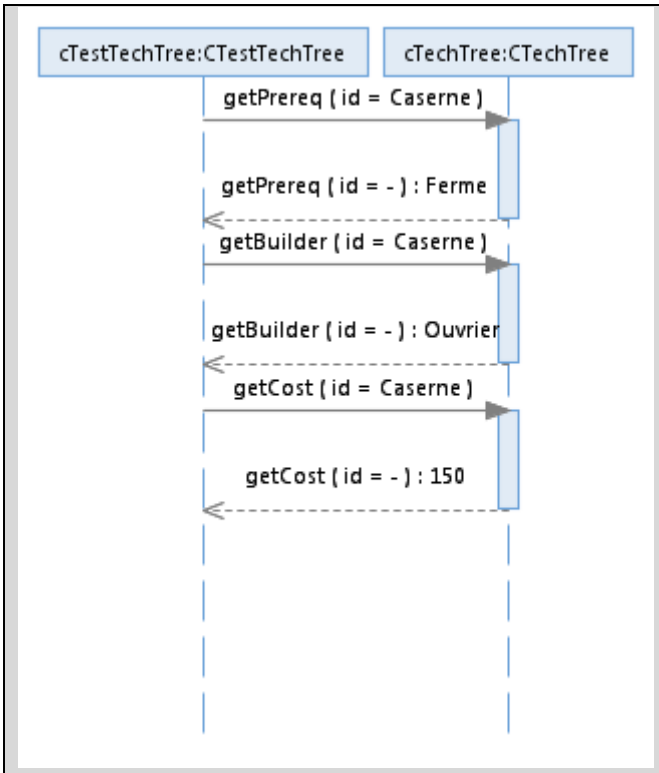
Binaire, en particulier 0 si on définit un bouchon

3. (0,75 point) Sur un diagramme de structure interne, représentez une configuration des composants pour le test.

Binaire

-20% si pas clairement des *instances* de composant.

4. (1 points) Sur un diagramme de séquence où vous utiliserez une ligne de vie par instance identifiée en 3), montrez les échanges nécessaires pour découvrir que la construction d'une caserne nécessite une ferme (prérequis technologique de caserne) et un ouvrier (pour construire la caserne) et coûte 150 PO. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.



10% lignes de vie

30% par invocation avec argument et valeur de retour correctes. On veut voir les trois invocations citées ici (avec ces paramètres et valeurs de retour).

S'il y a d'autres invocations en plus (genre interroger aussi sur l'ouvrier, ou sur la ferme) qui semblent modéliser un processus récursif on ne sanctionne pas, mais ce n'était pas demandé.

5. (1 point) Expliquez comment écrire en pratique le testeur et un test d'intégration qui couvre ce scenario, en supposant l'existence d'une *factory* statique **ComposantFactory** pour la construction des composants.

On utilise e.g. JUnit, on assertEquals des résultats attendus.

```
ITechTree tech = CompFactory.createTechTree() ;
assertEquals(Ferme, tech.getPrereq(Caserne)) ;
assertEquals(Ouvrier, tech.getBuilder(Caserne)) ;
assertEquals(150, tech.getCost(Caserne)) ;
```

Barème :

30% on cite JUnit/ on propose une réponse qui est clairement au niveau « code ».

30% le code instancie le composant à l'aide la factory

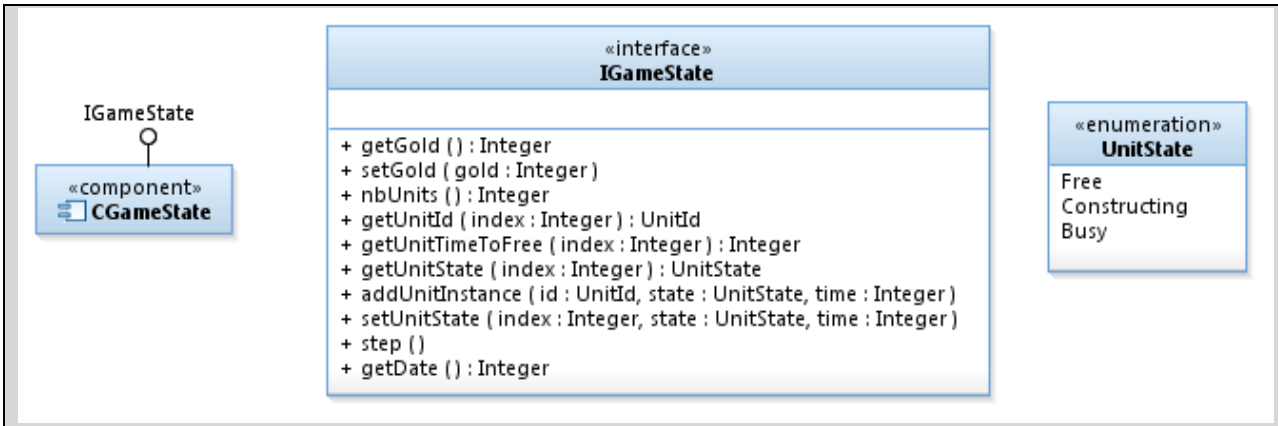
40% le code contient des assert

II. *Composant CGameState (5,5 points)*

On souhaite à présent construire un composant représentant l'état d'une partie. L'état d'une partie se résume à une date (en secondes depuis le début de la partie), à la quantité d'or (PO) disponible à cet instant et aux instances d'unité disponibles. Chaque instance d'unité est associée à son type, ainsi qu'à l'un des trois états « En construction », « Libre » ou « Occupé », et à un compteur qui décompte le temps restant avant que l'unité ne soit libre (0 si elle est déjà libre).

1. (2,25 points) Sur un diagramme de composants, représentez un composant bout de chaîne *CGameState* et son interface offerte *IGameState* répondant à ce besoin (avec signatures complètes). L'API doit supporter :
   - La consultation et la mise à jour du total d'or disponible de façon arbitraire,
   - La consultation de la date
   - L'ajout d'instances d'unités arbitraires (type, état, temps restant),
   - Accès en lecture : déterminer le nombre et l'état de chacune des instances d'unité
   - Accès en écriture : mettre à jour l'état et le temps restant pour une instance d'unité
   - Une opération *step()* qui laisse une seconde s'écouler et met à jour l'état des unités (-1 aux compteurs de temps restant des instances, retour à l'état libre si 0 atteint) et l'or disponible (+1 par ouvrier libre).

   Sans que ce soit imposé on suggère d'utiliser une API relativement basique, basée sur *size()* donnant le nombre d'instances et d'un accès indicé par l'index de l'instance à l'état des instances (indices entre 0 et size()-1).



10% le composant offre

10% accès r/w à gold

10% date

10% step

10% ajout d'unité avec 3 paramètres

Pour la suite, il faut qu'on voie clairement la multiplicité * (matérialisée par l'index dans le corrigé) pour avoir les points (sinon 0).

10% on peut dénombrer les unités

20% modification d'unité : on peut mettre à jour l'état et le temps restant d'une instance (ça peut être une seule ou deux méthodes)

20% consultation en lecture : on peut déterminer le type (10%) et l'état (10%) d'une instance. L'accès en lecture au temps restant n'est pas une faute mais ne compte pas de points.

-10% autres méthodes bizarres/non demandées.

Il y a moyen de faire une API qui passe un type d'unité en plus par rapport au corrigé, i.e. on obtient le nombre d'instances d'un type donné, on accède à la iéme instance d'un type donné… C'est correct aussi.

2. (1 point) On considère un composant **CInitialBuilder** chargé de créer l'état initial (une instance de *CGameState)* d'une partie : il offre une opération *createInitialState() :IGameState*. Représentez ce composant sur un diagramme de composants.

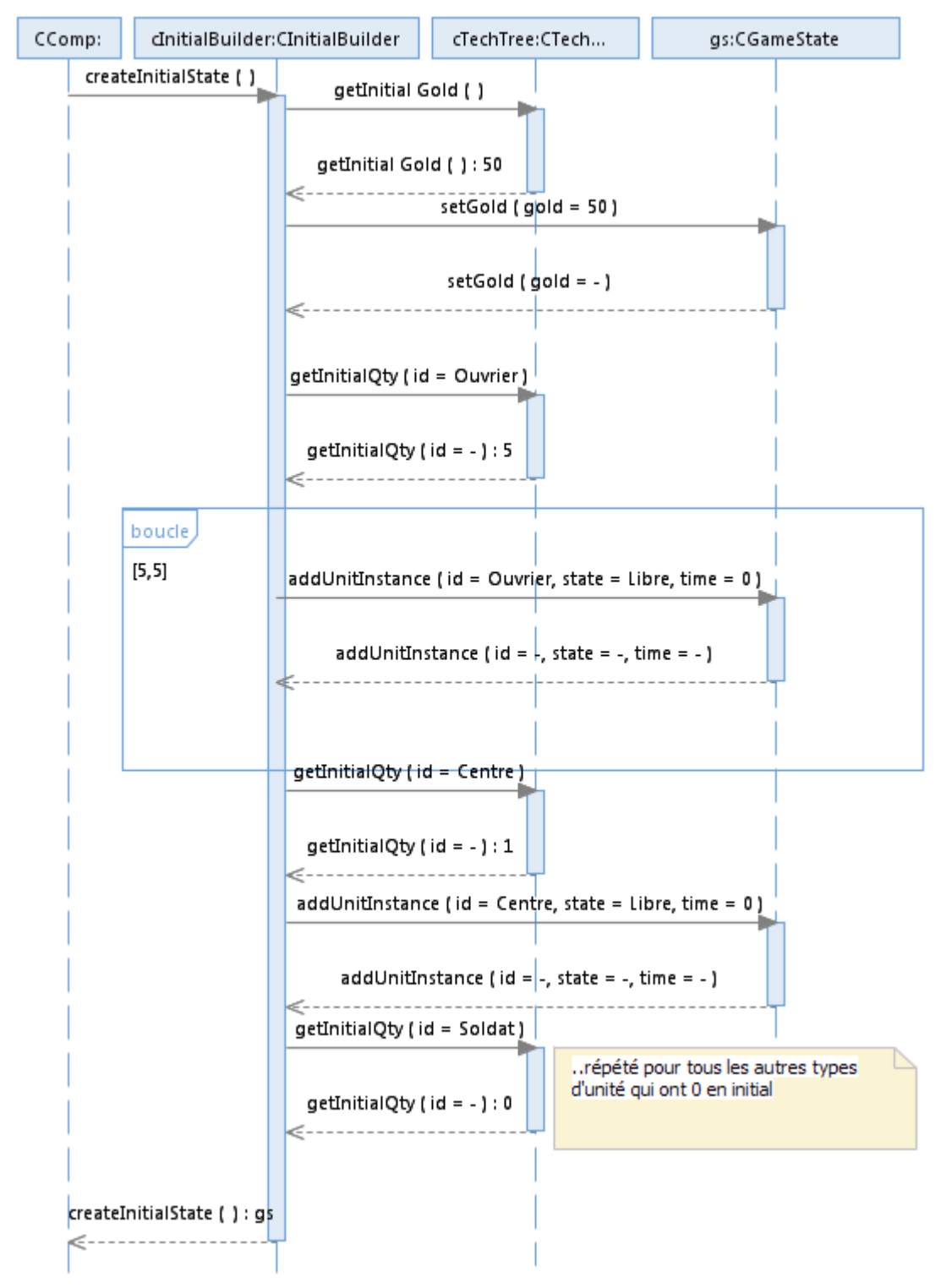
Barème :

20% offre

40% par dépendance

-30% par autre interface représentée.

3. (2,25 point) Sur un diagramme de séquence de niveau intégration, modélisez l'invocation de *createInitialState()* par un composant arbitraire *CComp* (jouant le rôle d'acteur). On suppose que l'état initial d'un *CGameState* est vide (aucune instance d'unité et zéro PO). On fera attention à être cohérent avec l'API proposée en 1) et celle de *ITechTree.* On utilisera des commentaires plutôt que de répéter des messages trop similaires. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.



Barème :

(sur 40% structure générale du dia est ok)

10% CComp invoque, on rend bien l'objet construit, une des lignes de vie de type CGameState.

30% initial builder fait au moins une lecture sur techtree, et au moins une écriture sur le gamestate

(sur 60% détails des invocations, avec valeurs correctes des paramètres)

10% màj du total d'or

20% on voit une itération sur toutes les unités possibles

10% on voit l'ajout du Centre

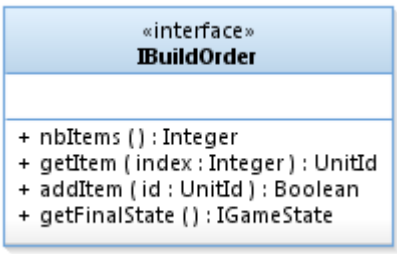
20% on voit passer les 5 ouvriers.

Pour avoir les points sur un message il faut source et destination correcte.

On donne la moitié des points si les arguments ne sont pas explicitement donnés.

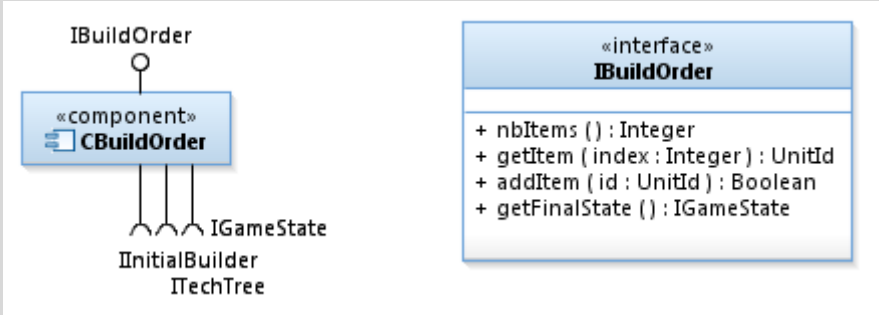
III. *Composant CBuildOrder (7,5 points)*

On considère à présent la représentation et la manipulation d'un *build order* (BO), c'est-à-dire une séquence ordonnée de constructions d'unités. L'objectif est de pouvoir mesurer le temps que prend un BO, c'est-à-dire la date à laquelle on arrive à exécuter la dernière instruction du BO.



Pour cela, le composant *CBuildOrder* démarre vide à partir de l'état initial du jeu (un *IGameState* cf. questions précédentes). L'invocation à « *addStep(id :UnitId) :Boolean* » vérifie que l'état actuel contient bien au moins une unité du type du « constructeur » et une unité du type « prérequis ». Si cette condition n'est pas satisfaite, l'opération rend *false*. Si les prérequis sont satisfaits, on fait progresser l'état du jeu (à l'aide de *step()*) jusqu'à ce que les ressources en or soient suffisantes pour acheter l'objet, qu'au moins une unité de type « constructeur » soit libre, et que le prérequis éventuel soit dans un état différent de « en construction ». La nouvelle unité est alors ajoutée à l'état courant, dans l'état « en construction », avec un temps d'attente égal à son temps de construction. On mobilise également une instance qui est le « constructeur » de l'unité, qui passe à l'état « Occupé » pendant le temps de construction.

1. (1 point) Sur un diagramme de composants, représentez ce composant *CBuildOrder*.



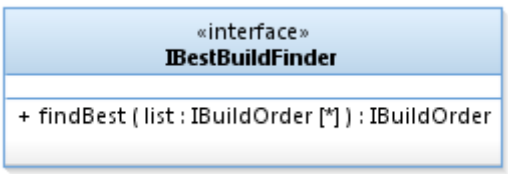
Barème

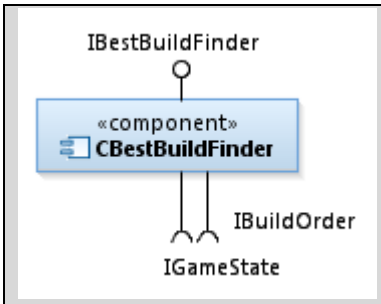
10% offre

30% par dépendance correctement identifiée

-30% par autre interface

2. (1 point) On souhaite comparer des BO qui ont été complétement construits. Dans un premier temps la métrique utilisée sera simplement la date de l'état final du BO. Représentez sur un diagramme de composants *CBestBuildFinder* qui offre ce service.





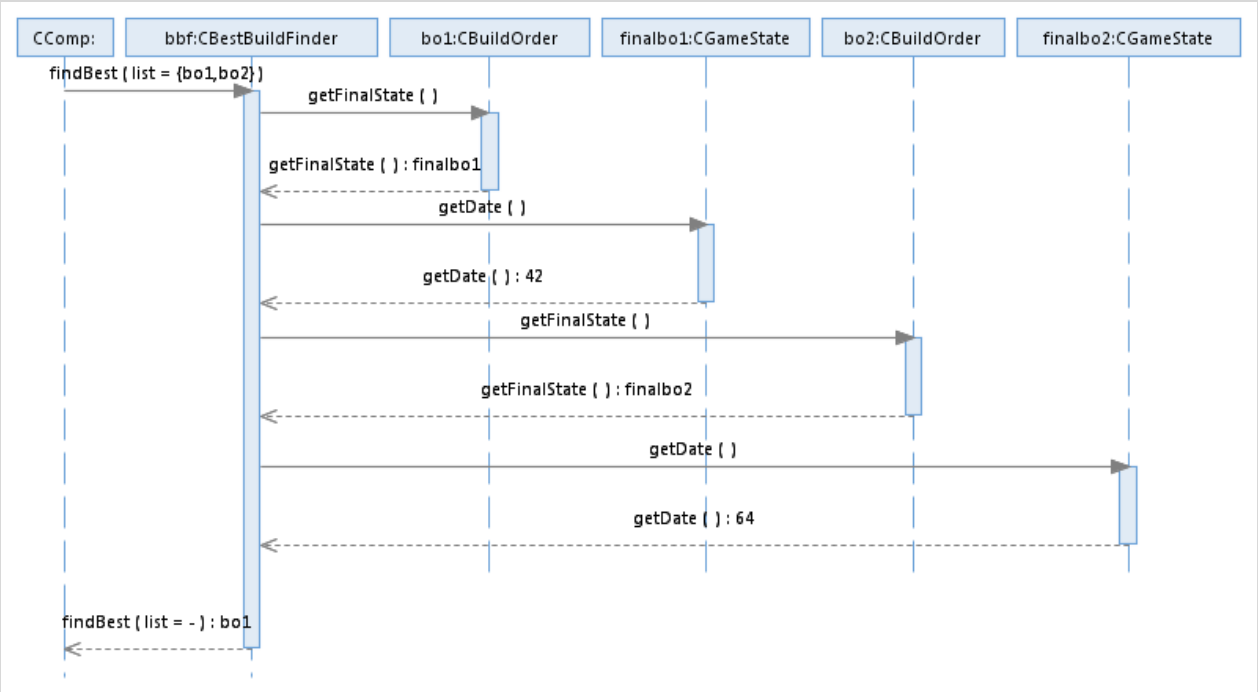
Barème :

20% offre

40% par dépendance identifiée

-30% par autre interface

3. (1,5 points) Sur un diagramme de séquence de niveau interaction entre composants, représenter un scenario où un composant arbitraire *CComp* (jouant le rôle d'un acteur) recherche le meilleur parmi deux BO à l'aide du composant *CBestBuildFinder*. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.



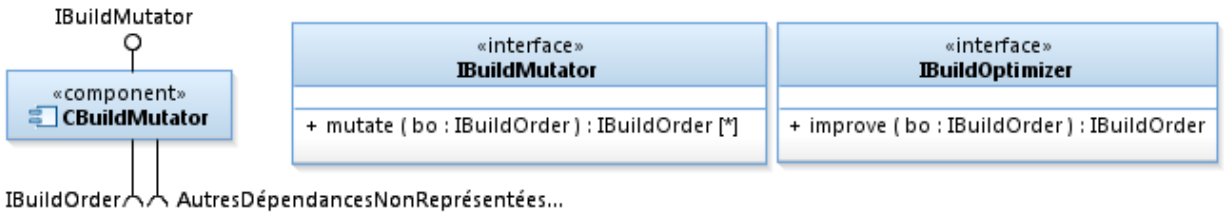
40% on voit bien apparaître 2 instances de BO et 2 instances de game state

20% les valeurs des messages sont congruentes aux noms des lignes de vie

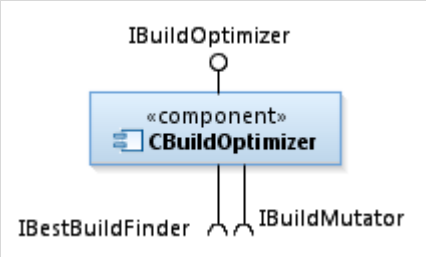
30% on voit getFinalState + getDate passer (pour chacun des bo)

10% on rend le plus petit des deux

4. (1 point) On vous fournit un composant *CBuildMutator* capable de proposer des variations autour d'un BO « de base ». Il offre l'interface *IBuildMutator* (cf. figure), et dépend d'autres composants qui ne seront pas détaillés dans cet énoncé.



En appui sur les composants développés dans cette partie III, on veut réaliser un composant d'optimisation *CBuildOptimizer* satisfaisant l'interface *IBuildOptimizer*. Représentez ce composant *CBuildOptimizer* sur un diagramme de composants.



Barème :

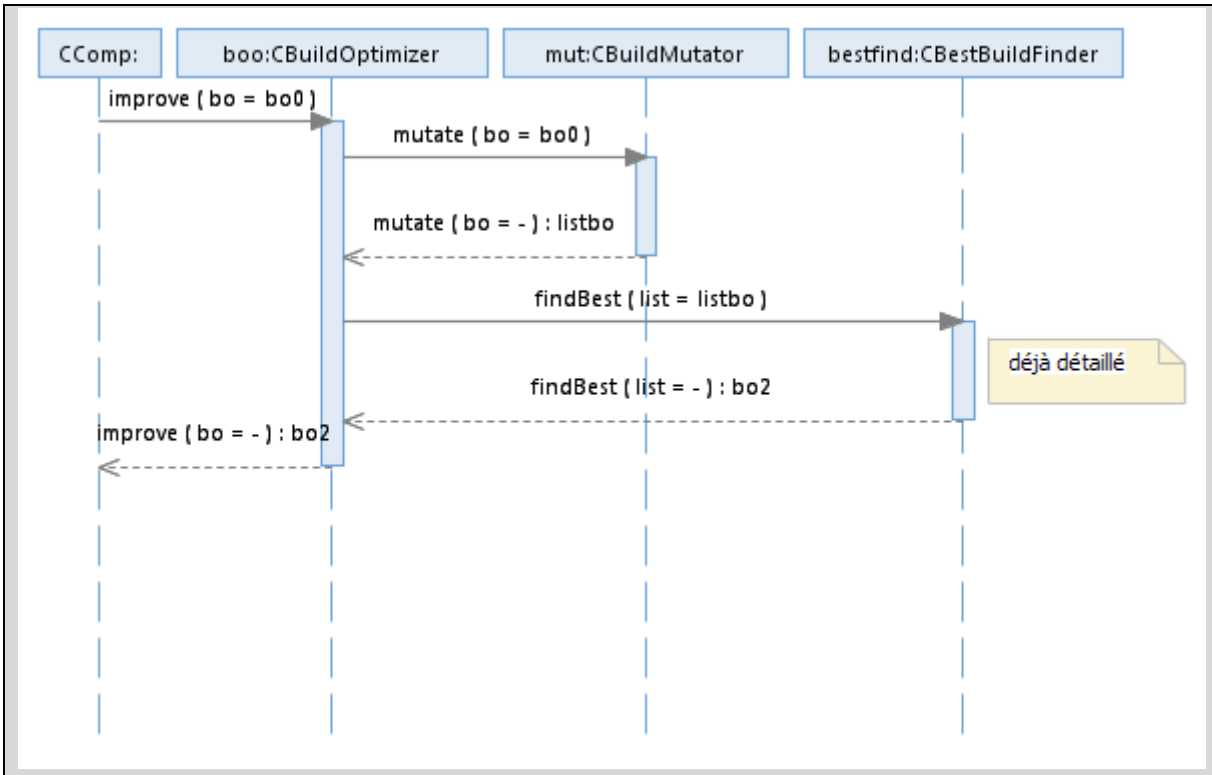
20% offre

40% par dépendance

-20% pour dépendance à IBuildOrder, en fait ce n'est pas utile

-30% par autre interface citée

5. (1,25 point) Sur un diagramme de séquence de niveau interaction entre composants, représenter un scenario où un composant arbitraire *CComp* (jouant le rôle d'un acteur) traite l'optimisation d'un BO donné. On ne détaillera pas les interactions qui ont déjà été décrites dans d'autres séquences de cet énoncé. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.



Barème

20% CComp invoque avec un bo, récupère un bo a priori différent

40% on voit passer mutate

40% on passe la liste à bestfinder

6. (1,75 point) Pour chacun des composants suivants, dites combien d'instances du composant seront présentes dans une configuration nominale du système SUBOO : CTechTree, CTesterTechTree, CTechTreeBouchon, CGameState, CInitialBuilder, CBuildOrder. Justifiez vos réponses succinctement.

Barème

Réponse non justifiée (juste un entier) = 5 % au lieu de 20%

20% : CTechTree : 1 seul, voire singleton. Important car représente la version du jeu + c'est read only donc pourquoi le dupliquer ? si on justifie « plusieurs versions du jeu » réponse N acceptée

20% : CTesterTechTree, CTechTreeBouchon : Aucun, on est "nominal"

20% : CBuildOrder : N instances : un CBO par build en cours d'évaluation,

20% : CGameState : N : au moins une instance de GameState par CBO

20% : CInitialBuilder : a priori un seul, voire en static/singleton.